HAND-HOLDING BASIC

Written By: NEIL BENNETT
In conjunction with Apple Computer, Inc.

SYSTEM AUTHOR'S ACKNOWLEDGEMENTS

It is a pleasure to recognize the contributions of those who have helped to bring Hand-Holding BASIC to its current state of development. I wish to express my indebtedness to Rudi Hoess, of Sydney, Australia for providing much guidance and encouragement; Roger Keating of Sydney, Australia for originating the structural concept of levels and giving much advise on the educational aspects of the system; Juris Reinfelds and Richard Miller, of the University of Woolongong, Australia, for contributing many valuable suggestions; and Portia Isaacson, of Dallas, Texas, for her encouragement.

I would also like to thank the SPECIAL DELIVERY SOFTWARE team at APPLE COMPUTER, INC. for helping to make the system into a product.

I am grateful to my wife Judy for her patience and understanding.

N. W. Bennett
September, 198Ø

FOREWORD

Hand-Holding BASIC is an implementation of the BASIC programming language designed specifically with the novice programmer in mind. Presented in four discrete units, it will hold the beginning programmer's hand through arithmetic expressions, the concept of variables, and the system-supplied functions. It will guarantee correct entry of program statements and expressions. It includes a full range of programming, program examination, and debugging.

Through the use of special screen displays and system commands, Hand-Holding BASIC puts much of its internal workings on public display. As a beginning programmer, you will have the opportunity to see just what the computer is doing every step of the way. If you've had some programming experience, but still are unsure just how programs work, Hand-Holding BASIC will help show you.

TABLE OF CONTENTS

CHAPTER 1
INTRODUCTION

The purpose of this manual is to teach the features of Hand-Holding BASIC. Chapter 1 describes the organization and scope of the manual; Chapter 2 presents an overview of Hand-Holding BASIC; Chapter 3 provides instructions on how to use the system; and Chapter 4 contains Appendices.

To run Hand-Holding BASIC, you will need an Apple II, one disk drive (16 sector), and a minimum of 48K of memory. Additional features are provided that use the game paddles; however, these are not absolutely required.

While this manual will teach you about Hand-Holding BASIC, it is not a tutorial on programming techniques. For that kind of information, you will have to refer to one of the many BASIC language programming tutorials readily available.

If you read Chapter 2 by itself, you will learn about the overall structure of Hand-Holding BASIC. If you read Chapter 3 by itself, you will learn the system functions and language statements. Chapters 2 and 3 are complementary. You should use them together.

You should also remember that learning to program a computer is, in a sense, like learning to play chess or bridge. You can't be expected to read a book of instructions and then sit down and play a flawless game, or write a flawless program. There are certain kinds of knowledge that come only from experience. You should, therefore, read until you get curious enough to experiment. Then experiment until you have to look back at this manual for an explanation.

While the system is rather unyielding on rules of language structure, there usually is more than one way to approach any problem. If you're curious, try it first one way and then another way. No programming book in the world is going to tell you exactly how to do everything. There is nearly always a better solution to any programming problem.

Perhaps the most important thing you can learn from your beginning programming experiences is that you can gain as much from your mistakes as you can from your successes. You should look carefully at all of the kinds of programming errors that Hand-Holding BASIC will detect and then deliberately set up situations to create each one of them. If you don't know how to make an error happen, then you may have a difficult time finding the cause of errors when they occur.

You'll never find out whether something will or won't work unless you try it. Experiment, experiment, experiment. Push your system to its limits. You can't break your computer from the keyboard.

This manual assumes that your Apple II system is correctly set up. If you're not sure that the system is ready to go, see Appendix A.

CHAPTER 2

ABOUT HAND-HOLDING BASIC

2.1 Scope Of This Chapter

Chapter 2 discusses the design of Hand-Holding BASIC. It does not provide instructions for using Hand-Holding BASIC. Its aim is to present the background information that will make the system more understandable. Chapter 3 will present instructions for using the system, ending with a demonstration program that includes experiments, and a sample problem for you to develop, modify, test, and adapt to your interests.

You should follow the suggestions in the introduction and take the time to diligently read Chapters 2 and 3 of this manual.

2.2 System Description

Hand-Holding BASIC is an implementation of the BASIC programming language designed specifically for beginning programmers. It has been written in conformance with the American National Standards Institute (ANSI) standard for minimal BASIC.

The elements of Hand-Holding BASIC are as follows:

```
   Numeric variables:  A-Z & AØ-Z9

   One-dimensional arrays

   Two-dimensional arrays

   String variables:  A$-Z$

   End :  END

   Implementation-supplied functions:

        ABS ATN COS EXP INT LOG RND SGN SIN SQR TAN

   Let :          LET var=exp

              LET str=strexp

              LET str1=str2
```

```
Control :   GOTO lnum

          IF  exp THEN lnum

          GOSUB lnum

          RETURN

          ON exp GOTO lnum,.....,lnum

          STOP

For    :    FOR var=expl TO exp2 STEP exp3

Next   :    NEXT var

Print  :    PRINT item p item p....p item

Input  :    INPUT var,....,var

Data   :    DATA datum,...,datum

Read   :    READ var,...,var

Restore :   RESTORE

Array  :    OPTION BASE n

          DIM var(n), DIM var (n,m)

Randomize:  RANDOMIZE

Remark :    REM remark-string
```

The programming skills that you learn using Hand-Holding BASIC will be directly transferable to other BASIC systems.

It is here that Hand-Holding BASIC's similarities to other versions of BASIC come to an end. The language set, the syntax, and the general computational capabilities are the same, but the environment in which it is housed is not. Hand-Holding BASIC makes its innermost workings accessible to the user in three ways: dynamic syntax checking; maintenance of six screen displays; and a special set of keywords that allow an unprecedented level of interaction. These three factors are defined in more detail as follows:

(1) Syntax Checking

The word "syntax" refers to the "grammar" of a computer language. Thus, when we use the word "syntax", the reader will know that we are referring to the rules of grammar for Hand- Holding BASIC.

The syntax of any program statement must be correct in order for it to execute. In the course of typing a statement you may attempt to enter an incorrect character. If so, your system will reject the faulty character. It will beep to let you know that you've made a mistake, and then it will give you another chance. If you attempt to enter a second incorrect character, the system will again reject it and beep. At this point it will display a list of those characters that are acceptable (called the select array). It will not allow you to proceed further until an acceptable character has been entered.

By doing syntax checking at the point of keyboard entry, Hand- Holding BASIC teaches you correct syntax with instantaneous feedback and precludes any syntax errors at execution time.

(2) Screen Displays

Every programmer dreams of the perfect program--the one that executes absolutely perfectly the first time. This rarely happens. Programmers must learn a set of diagnostic techniques along with their programming skills. These techniques usually involve such things as printing trace messages at certain program points, printing out intermediate values of variables, or setting breakpoints at suspected trouble spots.

Ultimately, an alert programmer will isolate and correct his bugs. But the traditional diagnostic approach has drawbacks. When a fault is isolated, the programmer normally notes the fault, halts execution, and studies the program listing to find out why the fault occurred. Hand-Holding BASIC will permit much of this diagnostic work to be done dynamically without interrupting program execution through the use of its six screen displays.

You can also switch from screen to screen by the use of special control characters. This will not affect the program's execution. You can halt execution at any time by pressing the space bar. Then restart it by pressing the carriage return (RETURN) key. You can switch from screen to screen while execution is halted. You can also adjust program execution speed from the keyboard or the Apple game paddles. You can single-step through a program.

1. The Command Screen is the one from which all normal keyboard entries are made: Programming, editing, and system commands. The Command Screen also includes dynamic information about which program step is being executed, and FORLOOP and subroutine activity.

2. The User Print Screen displays only program-directed print and input statements. Diagnostics occur while allowing the Output Screen to remain intact.

If you are not viewing the User Print Screen when an output is being printed, a flashing "O" will appear in the upper right hand portion of the screen you're watching to let you know about it. If you watch a program run to completion on the User Print Screen (when an END statement is executed), the display will automatically switch back to the Command Screen, and you will have to switch back to the User Print Screen to see the final lines of output.

3. The List-Trace Screen (LTRACE) displays the portion of the program being executed in source listing form. The next statement to be executed is highlighted. If you single-step through a program while watching the List-Trace Screen, you can watch how the program is executing.

4. The Chronological Trace Screen (CTRACE) displays a listing of the program statements as they are executed. It also shows the names of variables and their values as they are calculated. Single-stepping through a program while viewing the Chronological Trace Screen is another way of seeing exactly how a program is being executed.

5. The Monitor Screen displays a continuous record of the values of up to eight nominated variables and the program lines in which they attained those values. It shows the lines from which currently active subroutines have been called. The Monitor Screen will permit you to check variables without having to watch the Chronological Trace Screen.

6. The word "loop" in computer vocabulary refers to a group of program statements that are executed repeatedly. It is the repetitive aspect of this technique that gives rise to this term "loop" in the context of nearly all computer languages. Although the word LOOP is not a part of the Hand-Holding BASIC language, we will understand that the term FORLOOP does refer to the general use of the word "loop".

The FORLOOP Screen displays the FOR statement source line, the initial value of the index, the limit of the index, the index increment, and the current value of the index. Use of the FORLOOP Screen will allow you to check the values associated with a FORLOOP while the program is running, without separate diagnostic print statements.

(3) Keywords

There are a number of special commands in Hand-Holding BASIC called keywords, which give you more information about what's going on inside the system. These keywords will be discussed in more detail in the next section. Meanwhile, let an example suffice.

Suppose you were debugging a program in which the variable B8 was suspect. In traditional systems, it would be up to you to find the places where B8 was used in the program listing, and then to insert messages or breakpoints as you saw fit. In Hand-Holding BASIC, all you would have to do is to type the keyword BREAKFIND B8 and the system would find all occurrences of B8 and set breakpoints on them.

2.3 The Four Levels of Hand-Holding BASIC

Hand-Holding BASIC makes provisions for teaching programming by presenting its features in four levels: Level 1 teaches arithmetic syntax and allows the evaluation of arithmetic expressions. Level 2 introduces the concept of variables and the LET statement. Level 3 introduces the system mathematical functions. Level 4 provides computer statement, screens, and debugging tools.

Level 1

Level 1 teaches the proper use of arithmetic expressions. It allows the use of numbers, the four arithmetic functions (add, subtract multiply, and divide), and parentheses. Syntax checking takes place with each stroke of keyboard entry.

Level 1 will detect and report arithmetic errors of numeric overflow. In the case of an arithmetic error, you will be offered the opportunity to make use of the FINETRACE function. This function will display the computer's operations in evaluating an arithmetic expression to let you know at exactly what point the error occurred. FINETRACE may also be turned on under programmer control by simply typing in the word FINETRACE. Thereafter each expression will be evaluated step by step by pressing the spacebar for successive steps.

Level 2

Level 2 introduces the concept of variables and the assignment of values to variables using the LET statement. It allows the evaluation of arithmetic expressions using variables in accordance with the rules established in Level 1. Level 2 reinforces the fundamental rule that variables must be defined before they are used. This is done by reporting an undefined variable, if one is encountered, while evaluating an expression.

In Level 2, you will be able to use the keyword SYMBOLS (a Monitor Screen function). SYMBOLS will allow you to see which variables have been placed in the symbol table and their current values. The FINETRACE function is still available in Level 2. It shows how variables are replaced by their numeric values during the evaluation of expressions containing variables.

Level 3

Level 3 introduces exponentiation and mathematical functions. It allows the evaluation of expressions according to the rules established in Levels 1 and 2. It will also detect and report function errors in the case of function arguments that are out of bounds.

Level 4

Level 4 permits you to combine the lessons of Levels 1, 2, and 3 into real programming. It includes use of all the BASIC statements outlined in Section 2.2, as well as two special functions that allow use of the Apple game paddles: PDL (paddle) and BTN (button). Level 4 introduces a large set of keywords that pertain to programming, debugging, saving, and retrieving your programs.

These keywords are summarized as follows:

All of the following keywords pertain to the setting or turning off of breakpoints in various ways. The term 'lnum' refers to a line number.

```
BREAK lnum
BREAKFIND lnum
BREAKFIND var
BREAKFIND var=
BREAKLIST
NOBREAK
NOBREAK lnum
```

Delete individual statements or groups of statements from a program:

```
DEL lnum
DEL lnum1,lnum2
```

Edit a specific statement:

```
EDIT lnum
```

Find program statements containing the referenced items:

```
FIND lnum
FIND var
FIND var=
```

Set and remove variables to be displayed on the Monitor Screen:

```
MONITOR var
NOMONITOR
NOMONITOR var
```

List all variables specified for monitoring:

```
MONITORLIST
```

Clear the program:

```
NEW
```

Adjust program speed either from the keybord or by use of a game paddle:

```
PACE=n
PACE=PDL(n)
```

Save programs on a disk and later retrieve them:

```
ROLLOUT name
ROLLIN name
```

Begin program execution:

```
RUN
```

Level 4 gives you access to all six screen displays. Control characters let you switch from screen to screen. A special set of commands will allow you to halt (and restore) maintenance of four of the six screens in order to speed up program execution. Special commands allow you to set program speed at single-step or full speed during execution.

When the keyword RUN is entered, Hand-Holding BASIC checks the program for static errors before execution begins. Static errors include branching to non-existent lines, improper FORLOOP structure, and multiple or missing END statements. Static errors will be reported with a diagnostic message, and execution will not take place.

Level 4 also checks for certain kinds of errors which can be detected only during execution. If one of these is encountered, it will be reported with a diagnostic message, and execution will be halted.

CHAPTER 3

USING HAND-HOLDING BASIC

3.1 Scope of This Chapter

Chapter 3 is intended to help you learn the unique features of Hand-Holding BASIC. It is not intended to be a tutorial on programming practice. To learn BASIC programming, you should refer to one of the many beginning BASIC tutorials that are available.

Hand-Holding BASIC was written in accordance with the American National Standard for Minimal BASIC, ANSI publication X3.6Ø-1978. There are inevitably minor variations from one version of BASIC to another. A summary list of the functions and commands of Hand-Holding BASIC can be found in Chapter 4, Appendix B.

Brief System Description

Hand-Holding BASIC is built of the elements prescribed as standard for BASIC. Once you have learned how to program using Hand-Holding BASIC, you should be readily able to proceed to Apple's Integer BASIC, Applesoft BASIC, or Apple /// Business BASIC and learn the enhancements that each has to offer.

There are two unique features of Hand-Holding BASIC. First, and at all levels, it will not permit you to enter a statement containing incorrect syntax. Secondly, at Level 4 Hand-Holding BASIC will let you see (painlessly) a number of the things that happen during the execution of a computer program. It is a powerful debugging tool as well as an instructional aid.

Hand-Holding BASIC is presented in four levels, as follows:

Level 1: Permits the evaluation of arithmetic expressions only, including the use of parentheses.

Level 2: Includes all of Level 1; introduces the concept of variables.

Level 3: Includes all of Levels 1 and 2; introduces system-defined functions.

Level 4: Includes all of Levels 1, 2, and 3; permits the full range of programming, screen viewing options, editing, and debugging tools.

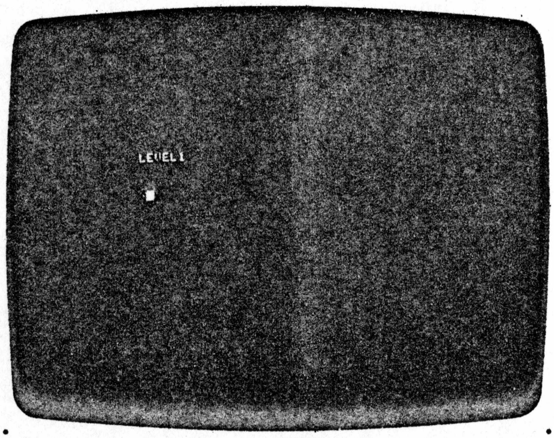
In addition to the normal BASIC commands, Hand-Holding BASIC includes a number of control commands called keywords. Each level allows all the options contained in lower levels. Therefore these keywords are introduced and explained once, and only once, where they first occur.

Even if you feel familiar with the BASIC language and method of arithmetic evaluation, you should still look at the sections describing Levels 1, 2, and 3 to be sure that you recognize the keywords and understand what they do.

Loading Hand-Holding BASIC Into Your System

To load Hand-Holding BASIC into your Apple II, insert the system disk in drive 1 and turn the power on. Your Autoboot ROM should load the operating system and program without further action on your part. (If you have trouble, refer to Appendix A).

The first screen you see should contain the name of the program and the copyright notice, etc. When that appears, press the space bar and your screen should look like the one shown below.



```
LEVEL 1
```

FIGURE 1. Level 1 Screen.

Now you're ready to proceed with Level 1.

3.2 Level 1

Scope of Level 1

Level 1 instructs you in the method of the computer's evaluation of arithmetic expressions. It enforces the rules of arithmetic syntax and introduces the associated Hand-Holding BASIC keywords.

Carriage Returns

After you type in a statement or a command, you should always press the RETURN key to say to the the computer, "now it's your turn". We will use the symbol<CR> to refer to the RETURN key. This is also called a "carriage return" upon occasion. Therefore, remember that whenever you see the symbol<CR> on the screen or in the manual, we are referring to the RETURN key.

Keywords

The keywords introduced with Level 1 (and permissible with all other levels) are:

1. SELECT. The select function displays (above the line being entered) those characters that are valid to be typed at the current point of entry. SELECT will be turned on automatically whenever two consecutive syntax errors are made and turned off automatically when the next correct entry is made.

   It may also be turned on manually by entering SELECT<CR> whenever the cursor is at its leftmost position, and will remain on until turned off by entering NOSELECT<CR> similarly. If the select function is turned on manually, the select array will be presented with each keystroke whether or not an error has been made.

2. NOSELECT. Turns off the select function.

3. FINETRACE. The FINETRACE function displays a series of lines that show the steps taken to evaluate the expression you have entered. One step is displayed each time you press the space bar.

   If you enter an expression that produces an arithmetic error, you will be offered the opportunity to have the FINETRACE function show you where the error occurred. You may accept that opportunity by pressing <CR>; otherwise, just continue as you wish. FINETRACE may also be turned on by entering FINETRACE<CR> whenever the cursor is at its leftmost position, and will remain on until turned off by entering NOFINETRACE<CR> similarly.

4. NOFINETRACE. Turns off the FINETRACE function.

5. LEVELn. Entering LEVELn<CR> takes the system to the Hand- Holding BASIC level specified by n (in the case of Level 1, n can be 2, 3, or 4).

Control Characters

Control characters available in Level 1 and their functions are as follows:

CTRL-X cancels the line currently on display and returns the screen to the condition it was in before the line was cancelled.

CTRL-I opens a space for you to insert new characters at the position of the cursor. It moves all the characters at or to the right of the cursor, further to the right for each new character you insert.

CTRL-D deletes the character under the cursor and moves all the characters on the right of the cursor to the left one position respectively.

Arithmetic Syntax

Level 1 allows the entry of numbers, the four arithmetic functions (add, subtract, multiply, and divide), and parentheses. The general syntax for Level 1 is

operand operator operand...

where operand is considered to be a number, or an expression embedded in parentheses. An operator is one of the arithmetic symbols +,-,*, or /.

The exception to the operand-operator-operand rule is that the + or - symbols, when used following another operator, will be interpreted as an indication of signed arithmetic. Thus, while

6*/3

and

6-*3

are not permissible,

2++2

4+-2

6*-3

and

```
6/-3
```

are.

Numbers as operands (the definition of an operand will be expanded in Level 2) have a maximum permissible length of eight digits. Any number greater than 9999999 must be expressed in scientific notation (as a power of 1Ø). Thus, 1ØØØØØØØ (1 x 1Ø^7) must be expressed as, and will be reported as, 1E7.

The symbol "^" stands for exponentiation. This symbol is made on the Apple II keyboard by shift-N.

The dynamic range of the system is from -1.85E17 to 1.85E17. Numbers in the range from Ø to +- 1.85E-19 will be treated as Ø.

Parentheses may be nested up to seven levels deep, and arithmetic operations may be chained together to the right limit of the screen, such as

```
(2+(3*(5/7-(4+(9*(6-(7--8/2+1)))))))
```

When you enter the eighth character from the right hand side of the screen, the highlighted message END-8 will appear in the upper right corner of the display. If you continue to enter characters, that message will change to END-7, END-6, and so on, until it flashes AT END, after which no more characters may be added.

If your statement is incomplete (as in the case of imbalanced parantheses), you will have to backspace to delete the statement, or else use CTRL-X in order proceed any further.

Syntax Errors and Correction

Whenever you make a syntax error, your Apple will beep to let you know about it. If you then enter a correct character, you may proceed in a normal fashion.

If you make two syntax errors in a row, Hand-Holding BASIC will assume that you need help and it will turn on the SELECT function. To go on from that point, you must enter one of the characters presented in the SELECT ARRAY.

It would be well for you to experiment a bit to see how the syntax error and correction work. Try entering these expressions just as they appear below, then see if you can make them correct. Also, note the contents of the SELECT ARRAY each time it appears.

```
2+*4

3+(4*7 <CR>

3+4*7) <CR>
```

```
        2..

        1234567899
```

Arithmetic Errors

You've already learned that Hand-Holding BASIC will not permit you to make a syntax error. Correct syntax is not, however, a guarantee against arithmetic errors such as dividing by zero or calculating a result that's outside the numeric range of the system. For example, enter the expression:

```
        (3*5-(7*2))/(3Ø-(6*4)-(3*2))<CR>
```

The result should be:

```
        ARITHMETIC ERROR 1/Ø

        PRESS <CR> FOR SLOW MOTION REPLAY
```

The symbol "<CR>" means the RETURN key on the Apple keyboard.

This message indicates that you have attempted to divide by zero. Press <CR> and then the space bar two or three times. What you're seeing are the steps by which Hand-Holding BASIC evaluated the expression. The error will be reported in the step in which it occurred.

You can calculate a result outside the numeric range of the system (called an overflow condition) by dividing a very large number by a very small number or by multiplying very large numbers. It is also possible to achieve a numeric overflow through repetitive addition of numbers. In the case of an overflow, the errors will be reported as:

```
        ARITHMETIC ERROR exp
```

where exp will be the arithmetic operation that caused the overflow.

Order of Evaluation

As you might have guessed from watching the first FINETRACE example above, there are rules for evaluating arithmetic expressions. These are:

1. Evaluate parenthetical expressions: start with the innermost.

2. Perform signed arithmetic.

3. Multiply or divide where possible (left to right).

4. Add or subtract where possible (left to right).

An example of the order of evaluation will be shown in Level 2. However, it is important that you understand the order of evaluation. Confusion could result in puzzling answers in your later programming, expecially when numbers are stored into variables. For example, evaluate these four expressions using Level 1.

```
9*4/12*2

4*9/12*2

9*(4/12*2)

(9*4)/(12*2)
```

Obviously, the same mix of numbers and operations has given us a variety of answers.

In general, you can control the order of evaluation by using parentheses to group operations so that the computations occur according to the order you intend.

Look at the rules for the order of evaluation and see if you can figure out why you got the answers you did.

Making Corrections

Somewhere along the line you will inevitably make a mistake. Not a syntax error. You won't be able to make one of those, but you may make a typographical mistake such as inverting two numbers, putting in an extra digit, or perhaps omitting a parenthesis.

You can edit lines as you are entering them, or by typing the edit command (EDIT nnnn) where nnnn is the line number to be edited.

If you want to change a character, press the back arrow key until the cursor is over the incorrect character. Type the new character. It will replace the old one, and the cursor will move one place to the right. Use the forward arrow key to move the cursor back to where you want it.

If you want to delete a character, use the back arrow key to put the cursor over the character you want to delete, then press CTRL-D. The unwanted character will disappear, and the characters to the right of the cursor will move left one place. If you wish to delete a series of characters, move the cursor to the leftmost one, then press CTRL-D as many times as necessary. Alternatively, to delete rapidly, enter CTRL-D, then press the REPT key and hold it down. The system will delete as many characters as you want, until you lift your fingers.

Use the forward arrow key to move the cursor right again to the end of the line, since automatic syntax checking does not occur unless the cursor passes over each character in the statement.

If you want to insert characters, first keep in mind that they will be inserted at the location of the cursor. Move the cursor to the desired position and press CTRL-I. A blank space will appear at the the cursor, and the other characters will be are pushed 1 position right. Type the characters you want to insert,and use the forward arrow key to return the cursor to the end of the line.

Returning the cursor to the end of the line is important because Hand-Holding BASIC will enter (and guarantee correct syntax) only the characters that the cursor has passed over.

Make a few mistakes on purpose and experiment with the correction procedures until you feel comfortable with them.

3.3 Level 2

Scope of Level 2

Level 2 introduces the concept of variables and the LET definition statement. Level 2 permits the evaluation of expressions containing variables according to the rules established by Level 1. It reinforces the necessity of defining variables before they are used in expressions, and lets you see (through the use of the SYMBOLS table), a list of variables and their current values.

Keywords

The keywords introduced with Level 2 (and then permissible with Levels 3 and 4) are listed below. All keywords introduced in Level 1 remain permissible in Level 2.

The SYMBOLS keyword switches the Level 2 display from the COMMAND screen to the MONITOR screen and causes all defined variables and their current values to be displayed. To return to the Command Screen, press <CR>.

BASIC Statements

LET var=exp

The LET statement means "assign to the variable var the value of the expression exp." Var may be any single letter of the alphabet, A-Z, or a letter followed by a digit in the range 0-9: A0, A1, A2, ..., A9; B0,..., B9; ...; Z0-Z9. Exp may be a single number, a single variable, or a combination of numbers and variables that conform to the rules of arithmetic syntax.

LET str=strexp

This LET statement means "assign to the string variable str the contents of the string expression strexp". Str may be any letter of the alphabet followed by a dollar sign, A$-Z$. The dollar sign identifies the variable as a string variable, and A$ is as different from A or A1 as A0 is from Z9. A string expression is another string variable or a literal. "Literal" constants can be part of a string expression. A "literal" is any combination of alphanumeric characters except quotation marks. "Literal" constants must be enclosed in quotation marks.

Essentially, string variables mean words. A string variable can be all numbers, such as in the example of LET A$="1981", but you cannot perform arithmetic operations among string variables or between string and numerical variables.

LET string1=string2

This LET statement means "assign to the string variable string1 the alphanumeric contents presently stored in string variable string2." Both string1 and string2 must conform to the above definition of string.

Using the Statements

(1) LET var=exp

According to its definition, a LET statement of the type LET var=exp may look like any of these:

```
LET A=3

LET A=3+2

LET A=B

LET A=B+3

LET A=B+C

LET Z9=(3+2*(B/C))/7
```

The most important concept to grasp about the LET statement is that although it looks like an algebraic equation, it most definitely is not. What happens inside the computer is that the value of the expression on the right side of the equal sign is placed in the memory location reserved for the variable name on the left. So, while an algebraic equation such as X = X + 1 would be meaningless, you can have a LET statement that looks like this:

```
LET X=X+1
```

What this statement means is "let the value of X plus 1 be placed in the memory location reserved for X." This statement, often called a counter, is used time and again in programming. In order to understand how it works, you must understand the difference between an algebraic equation and the LET statement.

There are really only three rules you have to understand about LET statements:

1. You must put a blank space between the word LET and the variable name that follows it.

2. Whatever variables are on the right side of the equal sign must already have been defined (given a value).

3. Whatever is on the right side of the equal sign must follow the syntax rules explained in Level 1.

Try, for example (don't forget <CR>),

```
LET A=3
```

and then evaluate some expressions using A.

```
A+3

A+5

A*2
```

Now define another variable

```
LET B=2
```

and evaluate some expressions with it and A.

```
A+B

A+B+1

A+2*B

(A+B)*2
```

Part of the emphasis of Level 2 is to teach you that you must define variables before you use them in expressions. Try evaluating:

```
A+B+C
```

You should receive the message:

```
C NOT DEFINED, GIVE VALUE NOW

LET C=
```

This message means that the system didn't know what to do because it didn't have all the information it needed.

Now enter 1 for C and see what happens.

Next, redefine variables A and B with:

```
LET A=3

LET B=6
```

and evaluate the expression:

```
A+(2*3+(A*B)+B)+7
```

If you didn't make any mistakes, you should now see:

```
THE RESULT IS 4Ø
```

How did the computer get from the expression to the result? It followed exactly the same rules for the order of evaluation explained in Level 1, except that it substituted the variable values when they were needed. Turn on the FINETRACE function (look back to the instructions for Level 1 if you don't remember how), enter the expression above, and begin pressing the space bar. Your screen should develop like this:

```
A+(2*3+(3*B)+B)+7

A+(2*3+(3*6)+B)+7

A+(2*3+(18)+B)+7

A+(2*3+18+B)+7

A+(6+18+B)+7

A+(24+B)+7

A+(24+6)+7

A+(3Ø)+7

A+3Ø+7

3+3Ø+7

33+7

4Ø
```

```
THE RESULT IS 4Ø
```

Also, note how the parentheses were dropped when they were no longer necessary.

Now try:

```
A+(2*3+(A*D)+7
```

and the space bar.

You should have made it to the second step before you were asked to enter a value for D. Enter 6 and start pressing the space bar again.

Notice that the evaluation didn't resume from where you left off, but instead went back to the beginning. This will happen every time you're in FINETRACE and you've forgotten to define a variable.

Don't forget to turn off the FINETRACE before going on.

(2) LET str=strexp

According to its definition, a LET statement of the type

```
LET str=literal
```

may look like any of these:

```
LET A$="MY NAME"

LET A$="1981"

LET P$="PART NO. L3777A"

LET Z$="HIS NAME IS 'JOHN'"
```

A LET statement of the type

LET str=strexp, which is the only kind of string manipulation allowed in Hand-Holding BASIC, has only one form, such as:

```
LET A$=B$

or

LET M$=D$
```

There are three rules you have to follow with the string LET statements:

1. You must type a blank between the word LET and the string variable name that follows it.

2. A literal string to the right of the equal sign must be inside quotation marks.

3. A literal string itself may not contain quotation marks. You may use a single quote if you wish.

You won't be able to do anything with literals or string variables until you get to Level 4 where the PRINT statement is available but you can get some idea of how they work in Level 2. First, enter:

```
LET N$="MY NAME"
```

Now enter:

```
N$<CR>
```

and observe the result. Now enter:

```
LET H$="HIS NAME"
```

followed by:

```
H$<CR>
```

Now try:

```
LET H$=N$
```

and:

```
H$<CR>
```

Now you should be able to see the words:

```
MY NAME
```

The Symbol Table

From the little experiment with string variables, you may have guessed that you can find the value of a variable by entering its name and then <CR>. You can. Try it with A, B, and C.

There's an easier way. Type:

```
SYMBOLS<CR>
```

Now your screen should say MONITOR in the upper right and down the left side should be the list:

```
A   3
```

```
B   6

C   1

D   6

H$ "MY NAME"

N$ "MY NAME"
```

and at the bottom:

```
6/6
```

This screen tells you three things. First, it is a Monitor Screen. It tells you what's going on inside the computer. You can no longer define variables or evaluate expressions as you could with the Command Screen. Second, it gives you the Symbol Table, and the names and values of all the variables you've defined so far. Third, it tells you how many variables have been listed so far and how many you've defined. The first 6 means six variables listed; the second 6 means six variables defined. The computer will list eight variables at a time. For example, if you had defined 27 variables, the first page would say 8/27. Press the space bar to get to a new page, and it would say 16/27, and so on.

Any time you are looking at the Symbol Table on the Monitor Screen you can return to the Command Screen by pressing <CR>.

3.4 Level 3

Scope of Level 3

Level 3 introduces the mathematical operation of exponentiation and system functions. Level 3 permits you to select whether arguments of trigonometric functions will be expressed in radians or degrees.

Keywords

The keywords introduced in Level 3 (which are then permissible in Level 4) are listed below. All keywords introduced in Levels 1 and 2 remain permissible in Level 3.

DEGREES. Causes the system to treat the arguments of trigometric functions as expressed in degrees. Radians is the system default.

RADIANS. Returns the system to the state of treating arguments in units of radians. Radians is the system default.

Mathematical Operators

The character ^ (shift N) causes the exponentiation of an expression and the general form is A^B, which means "A raised to the power of B". Thus, 2^2=4, 2^.5=1.41421; 2^(1/3)=1.259919.

In the definitions of functions that follow, all the functions that accept arguments will be shown in the general form

FUNCTION (arg),

where "arg" represents the argument of the function. Arguments can be constants, variables, or expressions containing constants, variables, or other functions, as long as the expressions are consistent with the rules of syntax. If the system encounters an unacceptable value for the argument, the function will return the message FUNCTION ERROR, together with the function at fault.

Arithmetic Functions

ABS(arg) returns the absolute value of arg.

INT(arg) returns the largest integer no greater than arg. For example INT(3.14159) will be 3; INT(-3.14159) will be -4.

RND returns a pseudo-random fraction between 0.0 and 0.9999999; RND accepts no arguments. See also RANDOMIZE in the discussion of Level 4.

SGN(arg) returns an indication of the algebraic sign of arg.

SGN(arg) is -1 if arg is negative, 0 if arg is 0, and 1 if arg is positive.

SQR(arg) returns the square root of arg. The value of arg for sqr(arg) must be positive.

Trigonometric Functions

SIN(arg) returns the sine of the angle whose value is equal to arg.

COS(arg) returns the cosine of the angle whose value is equal to arg.

TAN(arg) returns the tangent of the angle whose value is equal to arg.

ATN(arg) returns the angle whose tangent is equal to the value of arg.

Remember, the system will assume the value of arg for SIN, COS, and TAN to be in radians, and will return ATN in radians, unless degrees have been specified.

Exponential/Logarithmic Functions

EXP(arg) returns the value of the constant e (2.71828...) raised to the arg power; that is, EXP(arg)=e^arg.

LOG(arg) returns the value of the natural logarithm of arg. The value of arg for LOG(arg) must be positive.

Special Apple II Functions

PDL(n) returns a number between Ø and 255 that reflects the setting on the Apple game paddle specified by n. Permissible values of n are Ø and 1.

BTN(n) returns a 1 or Ø, indicating whether or not the button on the Apple game paddle (specified by n) is depressed. It returns a 1 if it is, and Ø if it is not. Permissible values for n are Ø and 1.

RND is often used in computer simulations to determine the chance of an event occurring such as slot machine wheels, the dealing of cards, or the roll of dice. An expression such as:

```
LET D1=INT(6*RND+1)
```

could represent the roll of a die.

Uses of PDL and BTN will also be discussed briefly in Level 4. Meanwhile, to get some idea of how they work, you can (after you determine which paddle is which) enter

```
PDL(Ø)<CR>
```

a few times with the knob at different settings and see what numbers come back, and:

```
BTN(Ø)<CR>
```

Once with the button left alone, and once with it depressed.

The following are some examples of syntactically correct expressions using functions. These are only examples and by no means suggest limitations or "the only way to do it." You are, as always, encouraged to experiment to learn as much as you can about how this portion of Hand-Holding BASIC works.

```
LET C=SQR(A^2+B^2)

LET R1=(-B+SQR(B^2-4*A*C))/(2*A)
LET S=SQR(1-COS(Z2*P/180)^2)

LET Q2=Q1+4*EXP(Q0/2)

LET Z2=ATN(Y/X)
```

3.5 Scope of Level 4

Level 4 brings you other tools you'll need to write, debug, and execute complete programs in BASIC. These tools fall into three categories:

1. Additional keywords which allow you to alter specific portions of a program or to examine a program in specific detail.

2. Six screen displays provide different perspectives about what's going on during program execution.

3. The BASIC statements themselves.

First, let's define the Hand-Holding BASIC statements.

BASIC Statements

Each BASIC statement must have a line number. Line numbers are limited to a maximum of four digits. Leading zero's are not allowed. A blank space must follow the last digit in the general format:

```
nnnn STATEMENT
```

Once a statement number has been entered, you can press the space bar when the next line number is to be entered, and the system will automatically create a new line number which will be ten greater than the previous one.

Only one statement per line is permitted.

A statement cannot be longer than the right hand side of the screen.

You may also delete individual statements by entering the line number and <CR>. If you delete a line in this manner, you will receive a message:

```
nnnn DELETED
```

End Statement

The END statement causes program execution to halt and returns control to the system. After an END statement has been executed, program execution can be reinitiated by use of the keyword RUN. A program must contain one, and only one, END statement, and it must be the last statement in the program. (See also STOP.) When an END statement is executed, display automatically returns to the Command Screen.

Control Statements

GOTO lnum

The GOTO statement transfers program execution to the program line number specified by lnum. A line with the number lnum must exist. A GOTO statement may not transfer execution into a FORLOOP body. GOTO may also be written: GO TO.

```
GOTO 120

GO TO 999
```

IF exp THEN lnum

If the relationship expressed by exp is true, program execution is transferred to the program line numbered lnum. If it is false, execution passes to the next line in sequence. Legal relational operators are equal to (=), greater than or equal to (>=), less than or equal to (<=), greater than (>), less than (<), and not equal (<>). The line numbered lnum must exist. Execution cannot be transferred into a FORLOOP.

```
IF A=10 THEN 480

IF A2+B2<0 THEN 900

IF 2*A<=3*B THEN 200

IF A-B>.005 THEN 180
```

GOSUB lnum

The GOSUB statement transfers program execution to the subroutine beginning at the program line numbered lnum. A line with the number lnum must exist. Execution cannot be transferred into a FORLOOP. GOSUB may also be written: GO SUB.

RETURN

The RETURN statement passes control of program execution to the line immediately following the one from which the subroutine call (i.e. the GOSUB) was made. For every GOSUB, there should be a RETURN statement. Similarly, a RETURN statement may not be legally encountered in program execution without there having been a corresponding GOSUB statement.

```
        2ØØ STATEMENT

        (subroutine text)

        25Ø RETURN

        (program text)

        4ØØ GOSUB 2ØØ

        9ØØ GO SUB 1ØØ1

        (program text)

        1ØØ1 STATEMENT

        (subroutine text)

        1Ø5Ø RETURN
```

ON exp GOTO lnum,...,lnum

ON - GOTO transfers program execution to the statement numbered lnum whose position in the list of line numbers corresponds to the rounded integer evaluation of exp.

For example, in the statement ON X GOTO 1ØØ,2ØØ,3ØØ,4ØØ, if the value of X is 3, then execution will be transferred to line number 3ØØ. If exp is less than 1, or exp is greater than the number of line numbers in the list, program execution will be halted. The referenced lines must exist. Execution may not be transferred into the middle of a FORLOOP.

```
        ON A+1 GOTO 199,235,375,4ØØ

        ON 3*B+7 GOTO 26Ø,2ØØ,24Ø,28Ø
```

STOP

The STOP statement forces program execution to halt at the current line number, but permits execution to resume with the next statement in sequence by pressing <CR>. When program execution has been halted by a STOP statement, you may examine the values of variables and still resume execution with the next statement number. If you change the program itself, however, you will be able to reinitiate program execution only with the keyword RUN.

```
        46Ø STOP
```

FOR Var=Exp1 TO Exp2 STEP Exp3

```
            (body)
```

```
NEXT Var
```

The FORLOOP allows us to perform (repeatedly) the block of statements between the FOR statement and the NEXT statement.

Var is any numeric variable which is not an array element. Var is to be used as a counter variable, helping to control the proper number of repetitions required in the FORLOOP. Var will receive a starting value, and will be changed on each repetition of the FORLOOP.

Expl is the initial value to be assigned to the variable Var.

Exp2 is the final value (limiting value) to be assigned to the variable Var.

Exp3 is the amount by which var will be incremented on each repetition as Var changes from the value Expl to Exp2.

When the FOR statement is encountered, Var is assigned the initial value Expl. Execution proceeds to the next statement in sequence. When the statement NEXT is encountered, Var is incremented by the amount specified by Exp3. If Var is less than the limit quantity Exp2, execution returns to the statement following the FOR statement. If Var is now greater than the limit, the body of the FORLOOP will not be executed, and execution will pass to the statement following the NEXT-statement.

The increment variable Exp3 may be negative, in which case the comparisons described for a positive increment are just the reverse. If no increment is explicitly stated, it is assumed to be the value 1 (one). Execution may not be transferred into the body of a FORLOOP.

```
FOR I=1 TO 10

     (body)

NEXT I

FOR J=4 TO 12 STEP A3

     (body)

NEXT J

FOR K=67 TO 43 STEP -8

     (body)

NEXT K

FOR N1=A+3 TO 3*B STEP M+2

     (BODY)
```

```
        NEXT N1
```

PRINT Item P Item P ... P Item.

The Print statement causes the specified items to be printed to the User Print Screen. Item may be an expression, a tab specification, or a null. P may be either a semicolon or a comma.

If Item is a negative numeric expression, a minus sign will be placed in front of it. If Item is positive, a blank space will be printed.

If the print statement is not followed by a list of Items, the print cursor will move to the beginning of the next line (and a blank line will be printed on the screen).

If P is a semicolon, the print cursor types the output item, and then stops immediately after the item.

If P is a comma, the cursor is advanced to the beginning of the next print zone. (The comma provides an automatic tab function. The screen is divided into three print zones: two zones are 16 spaces wide and one is 8 spaces. The tab specification allows the print cursor to be placed at the specified column.

Calculated tab specifications are rounded rather than truncated, and a calculated tab specification less than 1 will be set to 1. The maximum value for a tab specification is 255. Values larger than 40 will be reduced modulo 40.

```
        PRINT

        PRINT (A2+B2);C

        PRINT TAB(3);"ERROR"

        PRINT X

        PRINT A,B,C

        PRINT TAB(Y+Z/3);Y

        PRINT "END OF MONTH BALANCE FOR";B$;"ACCOUNT IS";B

        PRINT TAB(3);X;TAB(13);TAB(23);Z
```

Input Statement

INPUT var,...,var

The input statement causes program execution to pause for data values to be entered from the keyboard. The data entered from the keyboard must correspond to the type specified by the input statement.

```
INPUT X

3.14159

INPUT X,N$,B(7)

3,"SMITH",-5

INPUT A,B,C

2.5,Ø,3.77E-12
```

Data, Read, and Restore Statements

DATA datum,...datum

The DATA statement supplies the values sought by a READ statement. The data in the DATA statement must correspond to the types specified by the READ statement.

READ var...,var

The READ statement causes data values specified in a data statement to be assigned to their corresponding variable names.

```
READ X,N$,A

DATA 3,"SMITH",-5

DATA "JAN","FEB","MAR"

READ A$,B$,C$

FOR I=1 TO 5

READ A(I)

NEXT I

DATA 2,4,6,8,1Ø
```

RESTORE

The restore statement allows data to be reread by setting the data pointer back to the first datum in the list.

Array Declarations

OPTION BASE n.

The OPTION statement determines whether array subscripts will have a lower limit of 0 or 1 as specified by n. The OPTION statement, if used, must be the first statement in a program.

DIM var(n),...,var(n,m)

The dimension statement establishes the upper limit of an array's subscripts. The maximum value for a one-dimensional array is 4095. For a two-dimensional array the limits are 255 x 255. No array variable may be redimensioned. If no option or dimension is specified, all arrays are assumed to have a lower limit of 0 and an upper limit of 10. You should note that array variables are distinct from other variables beginning with the same letter. Thus, A, A1, A$, and A(1) are all different variables, and all may be used in the same program without producing system or logical errors.

```
OPTION BASE 0

DIM A(50)

DIM B(15),C(12,12),D(25,18)
```

RANDOMIZE

Each time the program memory is cleared by use of the keyword NEW or program execution is initiated by use of the keyword RUN, the pseudorandom number generator is reset so that it will start with the same number and then proceed with the same sequence. While this is a useful debugging tool, it takes all the excitement out of situations where the unpredictability of random numbers is desired. The randomize statement causes the pseudorandom number generator to start at an unpredictable value, and is therefore useful in simulations.

```
10 RANDOMIZE
```

REM remark-string

The REMARK statement allows you to provide internal documentation for a program. Remark-string may be composed of any combination of alphanumeric characters. The REMARK statement does not affect program execution and is not printed on the user screen. It appears only in the program listing.

Special Functions: PDL and BTN

While the PDL and BTN functions were described with Level 3, no mention was made there of how they might be used in a Hand- Holding BASIC program since programming wasn´t available in Level 3. PDL and BTN are not elements of the ANSI minimal BASIC, but because game paddles are available for the Apple, the ability to use them has been included in Hand-Holding BASIC.

Keywords

The first set of keywords in Level 4 deals with breakpoints. A breakpoint is a debugging tool that allows you to halt program execution at a predetermined point. When execution halts at a breakpoint, it tells you that the program was just about to execute the referenced step. This is important since it helps you trace the flow of execution.

When execution is halted at a breakpoint, you may examine variables. As with the STOP statement, you may not change the program and then continue. If you change the program, you will have to restart execution with the keyword RUN.

Whenever program execution halts at a breakpoint, the message

```
lnum BREAKPOINT
```

will appear on the third line of the screen, where lnum is the line number at which execution has halted. To resume execution after a breakpoint, press <CR>. There is no limit to the number of breakpoints that may be set.

BREAK line

Sets a breakpoint at the line number specified by line.

BREAKFIND lnum

Sets a breakpoint at the lines which reference lnum lists those lines with lnum highlighted. For example, BREAKFIND 10 might result in a list such as:

```
50 GOTO 10

90 IF B3 THEN 10
```

BREAKFIND Var

Sets a breakpoint at the lines in which the variable Var appears and lists those lines with Var highlighted. For example, BREAKFIND B1 might result in a list such as:

```
   4Ø LET A1=B1+3

   1ØØ LET B1=BØ+2

   7Ø IF B1<C4/3 THEN 4ØØ
```

BREAKFIND Var=

Sets a breakpoint only at the lines in which the variable Var appears as a dependent variable (a variable on the left side of an equals sign, for instance) and lists those lines with var= highlighted. For example, BREAKFIND X= might result in a list such as:

```
   3Ø LET X=1

   9Ø LET X=B2*COS(2*P1/18Ø)
```

Note that the Var= list will also contain statements such as:

```
   135 NEXT X
```

What actually happens in the next statement is:

```
   LET X=X+increment
```

BREAKLIST

Lists all lines in which breakpoints are currently set.

NOBREAK

Turns off all breakpoints.

NOBREAK lnum

Turns off the breakpoint set at the line number lnum.

You have already learned that you can delete a single line of a program by entering its number and <CR>. These next two keywords give you more flexibility in deleting unwanted program statements.

DEL line

Deletes the line number indicated by line.

DEL line1,line2

Deletes all program lines from line 1 through line 2 inclusive.

A keyword is provided for you to edit a specific line number.

EDIT lnum

Causes the line numbered lnum to be displayed on the screen just as it appears in the program listing. You can then use CTRL-I and CTRL-D to insert or delete characters as described in Level 1.

If you decide that you don't want to edit a line after all or that you'd like to go back and start over, entering CTRL-X will restore the line to the condition it was in before you started editing. When you have finished editing a line, don't forget to return the cursor to the right end of the line. Remember: the system checks syntax only for those characters that the cursor passes over.

The next three keywords locate and list the indicated references, but they do not set a breakpoint.

FIND lnum

Finds the statements in which the line numbered lnum is referenced and lists them with the line highlighted. For example, FIND 10 might result in a list such as:

```
50 ON X+1 GOTO 60,40,10,50

110 IF Y2<100 THEN 10
```

FIND Var

Finds the statements in which the variable Var appears and lists them with Var highlighted. For example, FIND B3 might result in a list such as:

```
45 LET B3=SQA(A2+B2-C2)

830 LET Y2=2+Y1+B3

890 LET Z1=ATN(B3*Y/X))
```

FIND Var=

Finds only the statements in which the variable Var appears as the dependent variable and lists them with Var= highlighted. For example:

```
FIND N1=
```

might result in a list such as:

```
40 LET N1=N1+1

175 NEXT N1 (See BREAKFIND Var=.)
```

The LIST keywords allow you to examine the program you're working on.

LIST

The LIST keyword causes the entire program to be listed. If there are more program lines than can be displayed on one screen, the list will scroll until the last statement has been printed. You can stop and restart the scrolling by pressing the space bar.

LIST lnum

Causes the program line numbered lnum to be listed.

LIST line1,line2

Causes all program lines from line1 through line2, inclusive, to be listed.

The next three keywords are used in conjunction with the monitor screen. You will see the effect of them when you go through the demonstration program presented later is this section.

MONITOR Var

Causes the variable Var, its current value, and the line number in which it attained that value to be displayed in the format:

```
line var value
 20  a1  399.6
 30  z9  432.1
```

A maximum of eight variables may be specified for monitoring in this fashion. Array variables may not be monitored. (All variables, however, will be displayed with the Chronological Trace Screen and the Symbols Table.)

NOMONITOR

Turns off the monitor function for all variables.

NOMONITOR Var

Turns off the monitor function for the specified variable Var.

MONITORLIST

Lists all variables specified for monitoring.

In learning computer programming, you will often be working with short programs or experiments, which, when completed, will no longer be needed. The next keyword allows you to delete your work area.

NEW

Clears the user program and Symbol Table and resets the pseudorandom number generator. In effect, it gives you a "clean slate" just as you had when you first entered Level 4. While NEW saves you the trouble of having to reload the system every time you want to start a new program, be sure that you really want to use it before you do. Once you enter NEW, whatever was in the program memory is gone. The only way you'll be able to get it back is to key it in all over again.

One of the unique features of Hand-Holding BASIC is that it allows you to adjust the speed at which a program executes. This may be done either from the keyboard or through the use of the game paddles. (See also the discussion of the screens for other effects on program execution speed.)

PACE=N

Allows you to adjust program execution speed from the keyboard by entering N between 0 and 255. PACE=0 halts program execution and permits you to single-step through it by pressing the ESC key. The program will execute one line each time the ESC key is pressed.

PACE=255 sets full speed for the system. With all screens being maintained, PACE=75 will result in an execution speed of approximately one line per second. Additionally, you may (during program execution) set PACE=0 by pressing the back arrow (<--) key. Then use ESC to single step. You may also set PACE=255 by pressing the forward arrow key (-->).

PACE=PDL(P)

Allows you to adjust program execution speed by the use of game paddle P, where P may be either 0 or 1. As described in Level 3, the setting of the paddle will result in a pace from 0 to 255. If you have set the pace by the game paddle, you can change the pace during program execution just by turning the paddle. If pace is not 0, pressing the paddle button will halt execution. Releasing it will cause execution to resume. If pace is zero, you can use the button to single-step through the program. One line will be executed each time the button is pressed. During execution, pressing the "0" key will put the system into the PACE=PDL(0) state. Similarly, the "1" key will put the machine in the PACE=PDL(1) state.

More than likely you'll develop some programs you'll want to keep or you might want to save a program in its current condition. The next keywords allow you to save and recall program files to and from disk.

ROLLOUT Name

Permits you to save the program in memory to disk with the system in exactly its current status. Name may be from one to six characters in length. The first character must be a letter; subsequent characters can be either letters or numbers.

Examples of permissible program names are: A, DEMO, MYPRO, PROG99. If you have only one disk drive, entering ROLLOUT name will display the following message:

```
PLACE ROLLIN-ROLLOUT DISC IN DRIVE 1
PRESS SPACE TO CONTINUE
```

After your program has been stored on the disk, a second screen message will appear that says:

```
PLACE HHB SYSTEM DISC IN DRIVE 1
PRESS SPACE TO CONTINUE
```

If you have two disk drives, be sure that your program storage disk is in drive 2 and then type ROLLOUT name,2. When your program has been stored on the disk, you will be returned to the command screen.

Note: If you do use the drive 2 option, then drive 2 will become the system default for all other storage and retrieval operations. If you want to use drive 1 you will have to do so by entering ROLLOUT name,1.

ROLLIN name

Permits you to read the program name from disk back into the system. If you have only one disk drive, entering ROLLIN name will get you a screen message that says:

```
PLACE ROLLIN-ROLLOUT DISC IN DRIVE 1
PRESS SPACE TO CONTINUE
```

After your program has been read in from the disk, a second screen message will appear that says:

```
PLACE HHB SYSTEM DISC IN DRIVE 1

PRESS SPACE TO CONTINUE
```

If you have two disk drives, be sure that your program storage disk is in drive 2 and then type ROLLIN name,2. When your program has been read in from the disk, you will be returned to the Command Screen.

Note: If you do use the drive 2 option, then drive 2 will become the system default for all storage and retrieval operations, and if you want to use drive 1 only, you will have to do so by entering ROLLIN name,1.

When the program comes back in, the system will be in exactly the same condition it was in when you rolled the program out. All the screen selections will be the same, the breakpoints will be the same, the point of execution will be the same, and so on. If you were debugging, you'll be able to pick up right where you left off.

Finally, at some point, you're going to want to make a program start executing.

RUN

Causes two sets of actions to occur.

First:

1. The symbol table is cleared.

2. The pseudorandom number generator is reset.

3. The data pointer is reset to the first item.

4. All branching statements (GOTO, ON-GOTO, GOSUB, and IF-THEN) are checked to be sure their destinations exist.

5. All FOR-LOOPs are checked for proper structure.

6. All branching statements (except RETURN) are checked to make sure they do not transfer execution into the body of a FORLOOP.

7. The existence of only one END statement is assured.

Second, if the program passes all the tests, then RUN also causes program execution to begin.

It is important for you to know about items 1-3 so that you understand why your program executes as it does.

If any errors of the types described in items 4-7 are encountered, a diagnostic message will be issued and execution will not take place. These error messages are described in detail in the section "Static

Errors." There are other possible programming errors that cannot be detected until execution time. Messages resulting from those errors are described in detail in the section "Dynamic Errors."

Screens

Hand-Holding BASIC maintains information on six kinds of activity during program execution. It can put this information on display in six different screen formats at any time. These screen formats and the information they contain will be described in conjunction with the demonstration program. A reference summary of the screen descriptions and control keys is presented here for your convenience.

Additionally, you will be offered the ability to "halt" and restore the maintenance of four of the six screens at any time during program execution. These screens contain a great deal of information. They can show you exactly how your program is executing and serve as a powerful debugging tool. Maintaining all the special screen information is quite costly in terms of execution speed. At some point in the development of a program, you will no longer need to keep the special screens up-to-date. Stopping all screen maintenance can speed up your program execution by a factor of about four and a half. If you do decide that you want a special screen back, you can always restore its maintenance.

When you halt a screen, the halt number will appear highlighted under the name of the screen or display. When you restore a screen, the highlighted halt number will be removed.

Command Screen

This is the screen from which you'll do all your programming, editing, and other activities normally associated with the programming process. The Command Screen is maintained at all times and cannot be halted. If you are viewing another screen and wish to switch back to the Command Screen, press CTRL-C (if execution has paused for a system command, or just C if input is not being requested).

User Print Screen

This screen is analogous to a printer. It shows information that the program has determined to print, including usual outputs as well as other items such as prompts for input made during program execution. Whenever you run a program, the screen display automatically switches from command to user print, and to leave it you will have to use one of the screen control characters.

The User Print Screen is maintained at all times and cannot be halted. If you are viewing another screen and wish to switch back to the User Print Screen, press CTRL-E if execution has paused for a system command, or just E if input is not being requested. If you are not viewing the User Print Screen when an output is printed, a flashing "O" will appear in the upper right hand portion of the screen you're watching to let you know about it.

If you watch a program run to completion on the User Print Screen, when an END statement is executed, the display will automatically switch back to the Command Screen and you will have to switch again to the User Print Screen to see the final lines of output.

List-Trace Screen

This screen maintains a listing of the portion of the program that is currently being executed, with the next line to be executed highlighted. Single-stepping through a program in the list-trace mode will show you the sequence in which the program lines are executing.

This screen is a dynamic instructional aid as well as a debugging tool. If you are viewing another screen and wish to switch to the List-Trace Screen, enter CTRL-Z (if execution has paused for a system command), or just Z if input is not being requested. To halt maintenance of the List-Trace Screen, enter 2. To restore it, enter 3.

Chronological Trace Screen

This screen maintains a list of the program lines as they are executed, along with the name and value of any variable changed in a program line. It might be thought of as a printed record of the activity shown on the List-Trace Screen, and serves as an instructional and debugging aid in the same way. If you are viewing another screen and wish to switch to the chronological trace screen, enter CTRL-T if execution has paused for a system command or just T if input is not being requested. To halt maintenance of the Chronological Trace Screen, enter 4. To restore it, enter 5.

Monitor Screen

This screen shows a list of the program line numbers from which currently active subroutines were called, a record of all variables specified for monitoring by the keyword MONITOR var, their current values, and the program line numbers in which they attained their current values. If you are viewing another screen and wish to switch to the Monitor Screen, enter CTRL-Q if execution has paused for a system command or just Q if input is not being requested. To halt maintenance of the Monitor Screen, enter 6. To restore it, enter 7.

Caution: When you halt the Monitor Screen, it stops right where it is and keeps the information that was on display when it was halted. If you halt the Monitor Screen during subroutine processing and then restore it when the subroutines are no longer being used, it will display historical information which is no longer valid subroutine information.

FORLOOP Screen

This screen displays, for each currently active FORLOOP, a listing of the program line containing the FOR statement, the initial value of the index, the limit of the index, the increment, and the current value of the index. The FORLOOP screen is a particularly valuable debugging tool when using calculated loop indexes. If you are viewing a different screen and wish to switch to the FORLOOP Screen, press CTRL-F (if execution has paused for a system command) or just F if input is not being requested. To halt maintenance of the FORLOOP Screen, enter 8. To restore it, enter 9.

Caution: When you halt the FORLOOP Screen, it stops right where it is and saves the information that was on display when it was halted. If you halt the FORLOOP Screen during FORLOOP execution and then restore it when no FORLOOPs are being used, it will display historical, no longer valid FORLOOP information.

Summary Of HAND-HOLDING BASIC Screen Controls

Screen	Input	No Input	Halt	Restore
Command	CTRL-C	C		
User Print	CTRL-E	E		
L-Trace	CTRL-Z	Z	2	3
Chron Trace	CTRL-T	T	4	5
Monitor	CTRL-Q	Q	6	7
For-Loop	CTRL-F	F	8	9

Static Errors

In the definition of the keyword RUN, we learned how the system makes a series of checks on the program before execution begins. These are the messages that may be printed as a result of those checks.

ABOVE GOTO TARGETS ARE NOT DEFINED

This message means that you have tried to transfer program execution to a nonexistent line number. It will also list the lines in which the errors occurred with the faulty target line number highlighted, such as:

```
40 GOTO  291

90 ON N2 GOTO 140,180,40,300

75 GOSUB  3010

142 IF J/2<>INT(J/2) THEN  10
```

OPTION BASE MUST BE 1ST STATEMENT

If you have chosen to define the option base, you must do so as the first program line.

LISTED END STATEMENT SHOULD BE STOP

This message will also list the line where the erroneous END statement was found, such as:

```
    99 END
```

A program can contain only one END statement which must be the last physical line of the program. Any logical halts must be forced by a STOP statement.

END STATEMENT MISSING

A program must contain an END statement as its last physical line.

MORE THAN 5 DEEP FOR-LOOPS

For-loops can be nested only five deep.

INDEX REPEATED

The same index is repeated in two FOR statements without an intervening NEXT statement.

INDEX DOES NOT MATCH

This message means that the index variable in the listed next statement is not the same as the index variable in the FOR statement to which it should correspond. This message will list the line in which the error occured, such as:

```
    620 NEXT Z9
```

Remember that next statements have to be worked back from the inside out and make sure you don't have two of them backwards.

NEXT MISSING

This message means that there is no next statement to correspond with a FOR statement.

ABOVE TARGETS JUMP INTO FOR-LOOPS

This message means that you have tried to transfer program execution into the middle of a FORLOOP body, which is not permissible. It will also list the lines in which the errors occurred with the faulty line number highlighted as with the undefined GOTO targets.

Dynamic Errors

Dynamic errors are those errors that occur during program execution. When a dynamic error is encountered, execution will be terminated and the error type will be reported. The messages listed below describe the dynamic error conditions. Besides these full messages, the errors will also be reported in the upper left portion of the screen (in an abbreviated way) in the general format:

lnum MESSAGE

here lnum is the number of the line in which the error occurred. The abbreviated messages are shown in the descriptions that follow as (lnum MESSAGE).

If you look at the List-Trace Screen after an error has been reported, the highlighted line will be the one in which the error occurred.

If you look at the Chronological Trace Screen after an error has been reported, the last (lowest) line shown will be the one in which the error occurred.

ON INDEX nn IS OUT OF RANGE (lnum ON ERROR)

This message means that the index in an ON-GOTO statement is less than one or greater than the number of target lines in the list; nn is the bad index value.

READ-DATA MISMATCH (lnum READ-DATA MISMATCH)

This message means that a datum in a DATA statement was not the type specified in the READ statement. That is, an attempt was made to read a string value into a numerical variable, or vice versa. The mismatched variable name and datum will also be listed under the error message in a form like:

```
A1="ABCD" or

B$=7.85
```

READ IS OUT OF DATA (lnum OUT OF DATA)

This message means that a READ statement can find no more data items to read.

VARIABLE TABLE FULL (lnum VAR TAB OFLOW)

This message means that you've defined more variables than the Symbol Table can hold (255 maximum).

FOR-LOOPS SIX DEEP (lnum FOR-LOOP 6 DEEP)

This message means that you´ve nested FORLOOPs more than the allowable five deep

TAB (nnn) IS OUT OF RANGE (lnum TAB RANGE ERROR)

This message means that you have issued a tab call that is greater than 255; nnn is the bad tab value.

GOSUB STACK FULL (lnum GOSUB OFLOW)

This message means that you have issued more than twenty GOSUB calls without encountering a RETURN statement. While there is no limit to the number of subroutines you can have, the GOSUB stack can maintain only twenty RETURN targets. This error might occur when you have not placed a RETURN where you really intended it to be, or a subroutine is erroneously re-calling itself.

RETURN & GOSUB STACK EMPTY (lnum GOSUB UFLOW)

This message means that execution has encountered a RETURN statement without a corresponding GOSUB statement.

var NOT DEFINED (lnum UNDEFINED VAR)

This message indicates that the variable var was not found in the Symbol Table. Unlike Level 2, Level 4 does not permit you to assign a value for var and then continue. You have to go back and define var with a LET statement in your program and then run it again.

var( ) OUT OF RANGE (lnum ARRAY RANGE ERR)

This statement means that you have attempted to use an array subscript beyond the limits established by the OPTION statement or var´s dimension statement. The expression var( ) will appear in a form such as:

```
A(12)

B(88,

or

N(5,99)
```

where the last subscript shown is the bad one.

var HAS WRONG NUMBER OF SUBSCRIPTS (lnum WRONG # SUBSCRIPTS).

This message means that you've supplied either two subscripts for a one-dimensional array variable or only one subscript for a two-dimensional array variable.

STRING TABLE OVERFLOW (lnum STRING TABLE OFLOW)

This message means that you've tried to define too many string variables (6 maximum).

var HAS NOT BEEN DIM'ED (lnum ARRAY NOT DIM'ED)

This message means that you've used a subscript greater than 10 without first dimensioning the array.

ARRAY var IS DIM'ED TWICE (lnum TWICE DIM)

This message means that you have attempted to dimension var after it's already been dimensioned.

The Demonstration Program

A demonstration program has been included on your Hand-Holding BASIC disk to give you some experience with using the screen controls and Level 4 keywords, and to show you exactly what information each of the screens contain.

If you have just started reading here from Chapter 2 to run the demonstration program, put the Hand-Holding BASIC disk in drive 1 and then turn the power on. The program should load automatically without any further action on your part. (If you have trouble, refer to Appendix A.)

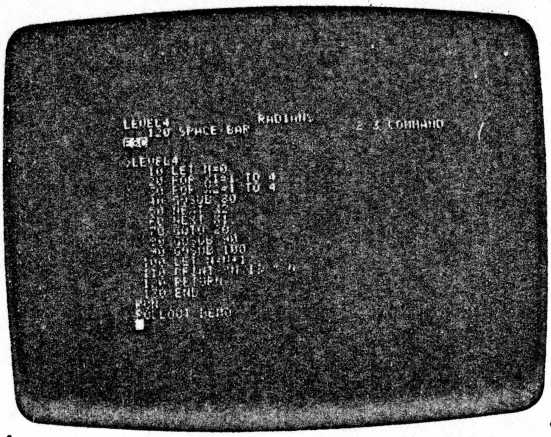
When the program has loaded, your screen should be displaying the program name, the copyright notice, and so on. Press the space bar, and the screen should be blank except for the words "LEVEL 1" in the upper left hand corner, with a flashing cursor below them.

The symbol '<CR>' will be used to mean "press the key marked RETURN on your Apple keyboard".

Type LEVEL4 <CR>, and the LEVEL 1 in the upper left corner should switch to LEVEL 4. Now you're all set to go on. Enter ROLLIN DEMO <CR>. You will get a screen message that says:

```
PLACE ROLLIN-ROLLOUT DISC IN DRIVE 1

PRESS SPACE TO CONTINUE
```



```
LEVEL4                        RADIANS            2 3 COMMAND
   120 SPACE BAR
E&C

>LEVEL4
    10 LET N=0
    20 FOR Q1=1 TO 4
    30 FOR Q2=1 TO 4
    40 GOSUB 80
    50 NEXT Q2
    60 NEXT Q1
    70 GOTO 20
    80 GOSUB 90
    90 GOSUB 100
   100 LET N=N+1
   110 PRINT "N IS " N
   1[illegible] RETURN
   120 END
  RUN
  SOLLOUT HERO
  █
```

FIGURE 2. Command Screen.

Since the demonstration program is on the system disk in drive 1, you can just press the space bar. After the program has been read from the disk, you will get the same message. Again, just press the space bar. Now you should be looking at a command screen with the program listing shown in Figure 2. The program itself is a do-nothing infinite counter whose sole purpose in life is to serve as a demonstration of Hand-Holding BASIC.

So that you'll understand what's happening when you single-step through it, here's how it works. Statement 1Ø defines the initial value of the variable N. The program then goes through statements 2Ø and 3Ø, where the indexes for the X1 and X2 FORLOOPs are set up. At statement 4Ø it calls a subroutine starting at statement 8Ø, which sends it to a subroutine starting at statement 9Ø. At statement 1ØØ, N is incremented by one and printed on the User Print Screen.

The first time the program encounters the RETURN statement at line 12Ø it will return to the target set by the most recent subroutine call, GOSUB 1ØØ in line 9Ø, clear that return target from the GOSUB stack, and land at line 1ØØ again. The second time it encounters the RETURN statement it will go to the next return target in the GOSUB stack, the one set by GOSUB in line 8Ø, which is line 9Ø). The third time it encounters the RETURN statement the return target will be the one set by line 9Ø, or line 1ØØ, and N gets incremented and printed again (and the return target set by line 9Ø is cleared). The fourth time the program encounters the RETURN statement, the only return target left on the GOSUB stack is the one set by GOSUB 8Ø in line 4Ø, or line 5Ø. Now, if all of this is correct, the sequence of events from line 4Ø should be (and you'll have a chance to check it out):

```
4Ø 8Ø 9Ø 1ØØ 11Ø 12Ø

1ØØ  11Ø  12Ø

9Ø  1ØØ 11Ø  12Ø

1ØØ  11Ø  12Ø

5Ø

4Ø  8Ø  9Ø  1ØØ  11Ø  12Ø

and so on.
```

After the X1 FORLOOP body of statements has been executed four times, the program goes back to line 2Ø, and the count goes on.

The information at the top of the screen is present on all screens except the User Print Screen. The message in the upper left corner tells you which Hand-Holding BASIC level you're in, and, in the middle of the top line, whether the system is treating angular values as being expressed in degrees or in radians.

At the left end of the second line, the number 12Ø indicates that program statement 12Ø will be the next one executed. When you see the term SPACE BAR this means that execution was halted by pressing the space bar.

At the right end of the second line, the number 2 indicates that the program is currently two deep in FORLOOP's, and the number 3 indicates that it is three deep in subroutines. The word COMMAND tells you that you're on the Command Screen. The other screen names will appear as LTRACE (list-trace), CTRACE (chronological trace), MONITOR, and FORLOOP.

At the far left on the third line, the highlighted letters ESC indicate that pace has been set to zero from the keyboard and that you can single-step through the program by pressing the ESC key. If the program were running at a pace less than 255, the numerical value of the pace would be shown in this area.

Now press CTRL-E to switch to the User Print Screen. There you'll see a column of outputs as shown in Figure 3. All outputs from program print statements appear on the user print screen, and there is no other information there so that you can have full use of the 4Ø-character by 24-line screen, just as you would with any other BASIC system.

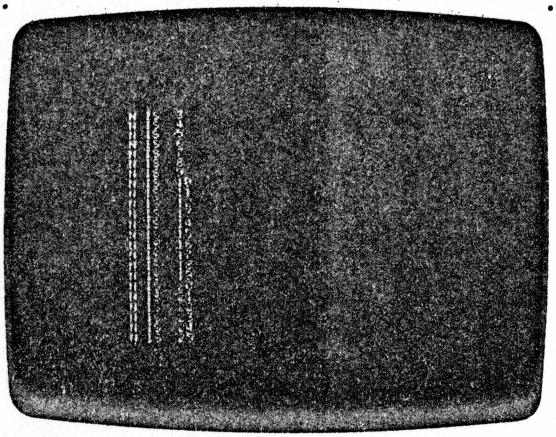


FIGURE 3. Output Screen.

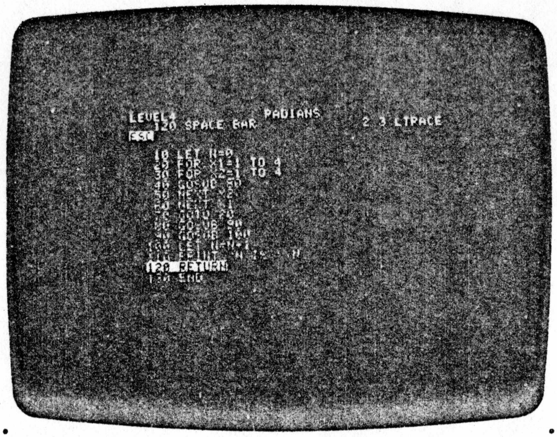
Press CTRL-Z to switch to the List Trace Screen. It should show, as in Figure 4, the same top-of-screen information described with the Command Screen, together with a listing of the program. But now line 12Ø is highlighted. This display of line in reverse video (black on white) indicates the next line to be executed. If you view the List-Trace Screen while a program is executing, you will see the highlight hopping around from statement to statement following execution. The screen display (the portion of the program shown) may change if the whole program is too long to appear on a single screen according to the following rules.

If the next statement to be executed is already on the screen, the highlight will simply move to it.

If the next statement is at the bottom of the screen, the program listing will scroll up, and the highlight will be on the second last line of the screen.

Otherwise, the next statement to be executed, together with the highlight, will appear at the top of the screen with succeeding program statements following.

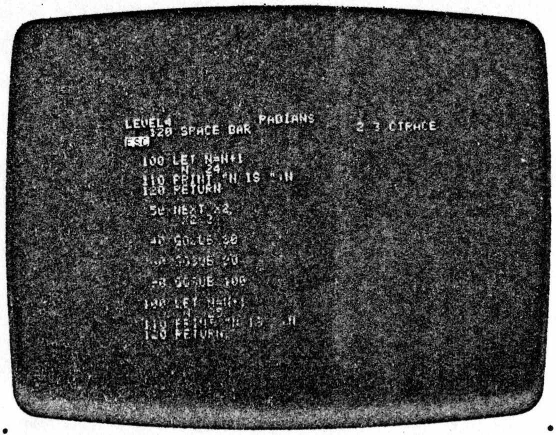
If you encounter a dynamic error, the highlighted statement will be the one in which the error occurred.



```
LEVEL4            RADIANS
   120 SPACE BAR                  2 3 LTRACE
ESC
   10 LET N=0
   20 FOR X1=1 TO 4
   30 FOR X2=1 TO 4
   40 GOSUB 80
   50 NEXT X2
   60 NEXT X1
   70 GOTO 20
   80 GOSUB 90
   90 GOSUB 100
  100 LET N=N+1
  110 PRINT "N IS ";N
  120 RETURN
  130 END
```

FIGURE 4. LTRACE Screen.

Now press CTRL-T to switch to the Chronological Trace Screen. There you will see, a mixture of program statements and variable names and values with line 12Ø at the bottom of the screen. The Chronological Trace Screen shows the next program statement to be executed. After execution, the name and value of any variable that was changed in that statement is also shown. Blank lines between statements mean that execution isn't sequential. If you encounter a dynamic error, the last statement shown on the Chronological Trace Screen will be the one in which the error occurred, as explained in the List Trace Screen.



```
LEVEL4                  RADIANS          2 3 CTRACE
  320 SPACE BAR
ESC
   100 LET N=N+1
         N  24
   110 PRINT "N IS ";N
   120 RETURN

    50 NEXT X2
         X2 3

    40 GOSUB 80

    80 GOSUB 90

    90 GOSUB 100

   100 LET N=N+1
         N  25
   110 PRINT "N IS ";N
   120 RETURN
```

FIGURE 5. CTRACE Screen.

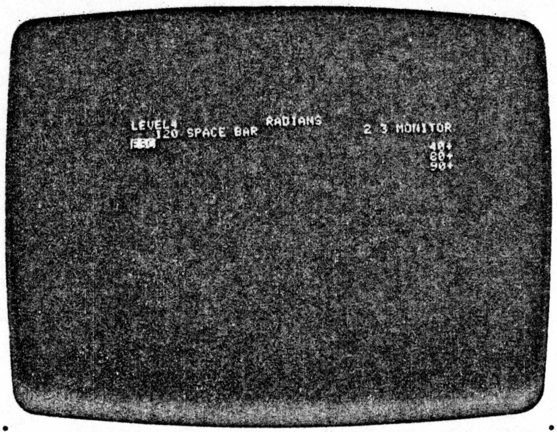
Now press CTRL-Q to switch to the monitor screen. In the upper right portion of the screen you should see the numbers:

```
40+

80+

90+
```

The number 3 next to the screen name means that you are three deep in subroutines. The numbers 40, 80, and 90 are the line numbers from which the subroutine calls were issued (a representation of the GOSUB stack). The "+" following each line number means that a RETURN statement will transfer execution back to the first statement following the one from which the subroutine call was issued.



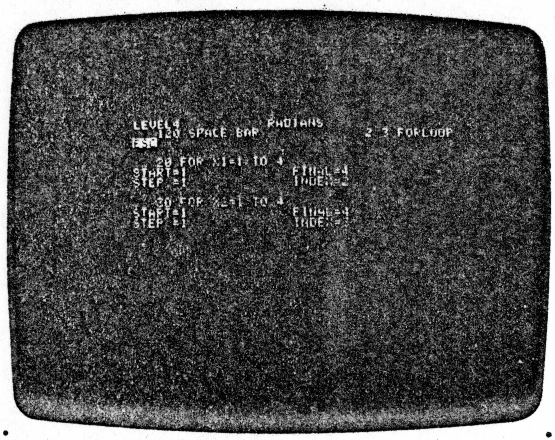
```
LEVEL4                  RADIANS
   120 SPACE BAR                      2 3 MONITOR
[illegible]                                   40+
                                             80+
                                             90+
```

FIGURE 6. Monitor Screen.

Now press CTRL-F to switch to the FORLOOP screen. There will appear, as shown in Figure 7, the two FOR statements plus two lines of information under each FOR statement. The number 2 to the left of the screen name means that you are two deep in FORLOOP's. These are the two FORLOOP's currently on the screen. For each FORLOOP, the information appearing below it is:

START	= the initial value of the FOR index
FINAL	= the limit of the FOR index
STEP	= the index increment
INDEX	= the current value of the index

The FORLOOP screen can be particularly valuable when you are using calculated indexes.



```
LEVEL4                 RADIANS
   120 SPACE BAR                          2 3 FORLOOP
ESC
     20 FOR X1=1 TO 4
START=1                      FINAL=4
STEP =1                      INDEX=2
     30 FOR X2=1 TO 4
START=1                      FINAL=4
STEP =1                      INDEX=1
```

FIGURE 7. FORLOOP Screen.

Experiments with the Demonstration Program

1. Press CTRL-C to return to the Command Screen. Then press <CR> to resume execution (which will automatically switch you to the User Print Screen) and set the pace at 255 by pressing the forward arrow key. One by one now, switch through the screens (using the appropriate letter--the CTRL key is not necessary when the program is executing). Watch what's happening on each of them. Make sure you stay on the MONITOR and FORLOOP screens long enough to see the GOSUB stack and FORLOOP information come and go a couple of times.

2. Return to the monitor screen and note that there isn't much on it except the GOSUB stack. Press the space bar to stop execution. From the Command Screen, enter:

   ```
   MONITOR N<CR>
   MONITOR X1<CR>
   MONITOR X2<CR>
   ```

   Now press <CR> one more time and watch what happens.

3. Stop execution again (by pressing the spacebar) and enter:

   ```
   PACE=Ø<CR>
   ```

```
RUN<CR>
```

Now switch to the List-Trace Screen. The highlight should be on line 1Ø. Press ESC three times to get to line 4Ø. Single-step through the program to check the action of the highlight against the program description. You should be sure you understand what's happening.

4. When you've single-stepped through the program as far as you like, return to pace 255 by pressing the forward arrow key (-->). Watch the highlight move until you have a good feeling for how fast the program is running. Now stop the maintenance of the Chronological Trace, Monitor, and the FORLOOP screens by pressing (not too rapidly) 4, 6, and then 8.

   Watch how the speed of the highlight increases when the maintenance of these screens is not being performed. Also note that as you stop maintenance of the screens, the control numbers appear highlighted underneath the screen name.

   You may, if you wish, restore maintenance of those screens by pressing 5,7, and 9, and watch execution speed slow down again. If you want the diagnostics, leave the screens on. If you want speed, you can turn them off.

   If you jumped to the demonstration program from Chapter 2, you should now go back and read the instructions. If not, then proceed with:

5. Stop execution again, then enter:

   ```
   LIST<CR>
   ```

   and you will see a new program listing come up on the screen (you can always get a listing in this way). Now enter:

   ```
   NEW<CR>
   ```

   and try listing the program again. It's gone! That's what the keyword NEW does. Also try running the program again. It won't work because you can't run a program that's not there.

   With the user program cleared, you're now ready to go on to the problem.

Love Thy Neighbor

Here's the problem: a number of points are distributed at random on a straight line. What is the probability that a point is it's nearest neighbor's nearest neighbor (in magnitude)? For example, if point 1 is the "nearest neighbor" to point 7, what is the probability that point 7 is point 1's nearest neighbor?

The situation is shown in the list of points below:

Point Number	Value
1	.1898498
2	.6146544
3	.Ø212Ø971
4	.1151428
5	.4426269
6	.755371
7	.149414
8	.837921
9	.8239133
1Ø	.7354735

Assuming that no computer were available, an empirical (experimental) solution to the problem could be developed using a pencil and paper and a book of random numbers. As you might imagine, the more numbers you choose (and hence the closer you got to a real solution) the more tedious this problem would be to solve manually.

Fortunately, a computer is available. Ten numbers will be used to outline a solution, and the problem will be modeled and the program developed in five steps.

1. Place ten numbers at random on a straight line.

2. Determine which point each point considers as its nearest neighbor.

3. Test the program so far.

4. Determine which pairs of points regard each other as mutual nearest neighbors.

5. Count and report the number of pairs of mutual nearest neighbors.

The following sequence will model placing ten random points between Ø and 1 on a straight line. Key it in.

```
10 DIM X(10)

20 FOR N=1 TO 10

30 LET X(N)=RND

40 NEXT N

50 END
```

This sequence just fills each of the ten array slots with a random number. Run it, and while it's running, look at the List- Trace and FORLOOP screens and watch what happens as it executes (you can run it more than once if you want to). When it's finished, use the keyword SYMBOLS to look at the symbol table and see what numbers have been generated and stored.

The next step is to determine which point any given point considers as its nearest neighbor. What determines a nearest neighbor is the smallest difference between the test number and all the other numbers. The method that will be used to find the smallest difference is sometimes called "swapping out", and, in general, it works like this:

1. Initialize an impossible value as the smallest difference.
2. Find the difference between two points (new difference) and compare it to the smallest difference.
3. If the new difference is smaller than the current smallest difference, then replace the smallest difference with the new difference, record which point produced the new smallest difference, and go to the next point.
4. If the new difference is not smaller than the current smallest distance, just go on to the next point.

The following sequence will swap out, searching for the smallest difference, and record the point number that caused the smallest difference. The variables are:

S = smallest difference

N = any point number

T1 = the point whose nearest neighbor is being sought

D = the difference between the test point and any other point

T2 = the point that caused the smallest difference
(the nearest neighbor)

```
8000 LET S=100
```

```
8010 FOR N=1 TO 10

8020 IF N=T1 THEN 8070

8030 LET D=ABS(X(T1)-(X(N))

8040 IF D>S THEN 8070

8050 LET S=D

8060 LET T2=N

8070 NEXT N
```

How this sequence functions compared to the general situation of swapping out might be expressed like this:

8000. Initialize an impossible smallest difference. Since you know that all the differences here are going to be less than one, a value of S=100 will fail the first test in statement 8040, and the first real difference will become the first smallest difference.

8010. Set up the for-loop counters to check all ten points.

8020. If the current point is the same as the test point, go on to the next point (8070); that is, don't check the test point against itself.

8030. Define D as the difference between the test point and the current point. The absolute difference is used because only magnitude, not direction, is important.

8040. See if the new difference is larger than the current smallest difference. If it is, go to the next point (8070); if not,

8050. Replace the old smallest difference with the new smallest difference.

8060. Define T2 as the point that caused the smallest difference (the nearest neighbor).

8070. Establish the next point (and go to 8020) or quit (and go on to the next statement).

Key those eight lines in, then make a subroutine out of them by adding:

```
8080 RETURN
```

Now it's time to test what you've entered. To do that, add these statements:

50 LET T1=5 (and notice that you're overwriting the previous 50 END)

```
60 GOSUB 8000
```

```
70 PRINT T2

80 STOP

9000 END
```

List the program out to be sure it's okay, then run it. As before, look at the list-trace and for-loop screens and watch the program execute (and, as before, you can run it more than once if you want to). According to statements 50-80, you're going to use subroutine 8000 to find point 5's nearest neighbor, then print out what it is and stop. When the program has finished execution, use Ctrl-E to go to the user print screen and see what point 5's nearest neighbor is. It should be 2.

If you got the right answer, you can modify the program slightly to allow you to check a few more points. (If you didn't get the right answer, you'd better list out your program and find out what's wrong.) Go back to the command screen and enter:

```
50 PRINT "INPUT T1"

55 INPUT T1

80 GOTO 50
```

Run the new program and try 5 again. Then try 2--its nearest neighbor is 10. Try 10--its nearest neighbor is 6. Try 6--its nearest neighbor is 10.

Out of the four observations you've made, two points out of four regard each other as nearest neighbors, which might tempt you to infer that chances are 50-50 that a point will be its nearest neighbor's nearest neighbor. Now you can alter the program a little more to test your hypothesis.

Break out of the input loop by entering a number and pressing the space bar before the program has finished executing. Delete line 55 with DEL 55, then enter these new lines.

```
11 DIM C(10)

50 FOR T1=1 TO 10

70 LET C(T1)=T2

80 PRINT T1,T2

90 NEXT T1

100 STOP
```

What this portion of the program will do is set up the test points (T1) with a for-loop rather than having you enter them from the keyboard, and then store all the nearest neighbors (T2) in the C array. Run it and watch the user print screen: the left column of numbers is the test points; the right column is the nearest neighbors. (Use SYMBOLS to have a look at all the variables you're maintaining now.) By observation, you could count up the mutual nearest neighbors--but why not go on to step five of the original plan and let the computer do the counting? Add:

```
100 LET Z=0

110 FOR T1=1 TO 10

120 LET T2=C(T1)

130 IF C(T2)<>T1 THEN 150

140 LET Z=Z+1

150 NEXT T1

160 PRINT Z

170 STOP
```

In this sequence, Z is used as a counter for the number of mutual neighbors. Statements 110-150 loop through all ten points; statements 120 and 130 check to see whether T1's nearest neighbor regards T1 as its nearest neighbor. If so, Z is incremented by one; if not, the next point is tried.

Now run the program and watch the results on the user print screen. At the bottom of the list of neighbors you should see the number of mutual nearest neighbors reported as 6, or, at least for this trial, it looks like chances are 6 in 10 of any number's being one of a pair of mutual nearest neighbors.

Finally, so that you aren't using the same numbers every time, add:

```
15 RANDOMIZE
```

Run the program as many times as you like and see what kind of answers you get.

On your own, you might want to try building a larger loop around such portions of this program as are necessary to get it to execute 10, 50, 100 or more times, and find the average number of mutual nearest neighbors for all runs. Or, refer to the June 1980 issue of OMNI magazine (page 108), from which this problem was adapted.

Happy computing.

CHAPTER 4
APPENDICES

APPENDIX A

SETTING UP THE APPLE II SYSTEM

This appendix includes a list of the equipment you´ll need to run Hand-Holding Basic on your Apple II. You do not need to read all the manuals, but they should be on hand to answer questions that may arise in operating the equipment (e.g., how to boot a diskette).

Hand-Holding BASIC is written in 6502 Machine Code. To use the program, you´ll need the following equipment:

- an Apple II with 48K bytes RAM; or
- an Apple II Plus with 48K bytes RAM and an Integer BASIC Firmware Card; or
- an Apple II Plus with the Apple Language System.

PLUS:

- an Apple Disk II with Controller (16-Sector PROMs) (two disks preferred);
- a Video Monitor or Television;
- game paddles (preferred, but not necessary).

For reference, you should have on hand a copy of the following manuals:

- This Manual (A User´s Guide to the Programs);
- an Apple II BASIC Programming Manual (Setting up the Apple II);
- DOS Manual (How to Boot the Diskettes).

Putting The Pieces Together

Here are the steps to follow to put your system together:

(1) To set up your Apple II, follow the instructions in the Apple II BASIC Programming Manual. You may not need to attach the Game Controllers, although there is no harm in doing so. Your Apple II must have at least the minimum amount of memory listed under the equipment description for you to use the programs.

(2) If you already have a Disk Operating System, and are using a version of DOS that runs in 13 sectors (DOS 3.2.1 or earlier), you will need to change two proms on your disk controller card to update your system to 16 sectors. Any version of DOS earlier than release 3.3 will need to be updated. These proms are also the same proms that come with the Pascal Language System. Consult a DOS 3.3 manual for these procedures.

Appendix B

NOTES REGARDING COPY PROTECTION

Special Delivery Software is copy protected, except in the case of program utilities or template-applications for major products (e.g. Apple PILOT).

In order to provide you with a backup capability, we enclose a second diskette. You should definitely store your backup in a safe location, and NOT use it. In the event your main diskette becomes damaged within the time period of Special Delivery Software's media warranty, you may return the main diskette to us for replacement, and continue to use your backup until we can send you a replacement.

Unlike most other copy protection schemes, our method is selective; protected and unprotected files may reside on the same disk. On this diskette, HANDHOLDING BASIC is protected, but the DEMO files are not. Also, program files you create from HANDHOLDING BASIC will not be protected. If you boot from DOS 3.3, you will be able to catalog this diskette. FID will enable you to transfer unprotected files between this diskette and other disks in either direction, provided of course, that there is sufficient space on the destination disk and it is not write-protected. In short, the unprotected files are normal DOS files with all the properties thereof. On the other hand, the protected files can only be run by booting from the disk on which they reside. Any attempt to copy, load, run, or verify them from normal DOS will result in an I/O ERROR. No damage is done; the fact is merely that normal DOS cannot access these files.

Memory protection is also in effect. If an unauthorized form of access is detected, the memory protection procedure will zero out all memory; this procedure is triggered by pressing the RESET key. Once triggered, your screen will be filled with R's (or Q's) on a white background. You will have to re-power your system to re-start any new application.

APPENDIX C

BASIC STATEMENTS USED IN

HAND-HOLDING BASIC

DATA
DIM
END
FOR-NEXT
GOSUB
GOTO
IF-THEN
INPUT
ON-GOTO
OPTION
PRINT
RANDOMIZE
READ
REM
RESTORE
RETURN
STOP

Functions: ABS ATN BTN COS EXP
INT LOG PDL RND SGN
SQR TAN

Variables: A-Z; A0-Z9
One-dimensional arrays
Two-dimensional arrays
A$-Z$

APPENDIX D

HAND-HOLDING BASIC KEYWORDS

```
BREAK line
BREAKFIND line
BREAKFIND var
BREAKFIND var=
BREAKLIST
   NO BREAK
   NO BREAK var
DEL line
DEL line1,line2
EDIT line
FINETRACE
   NOFINETRACE
FIND line
FIND var
FIND var=
LEVELn
MONITOR var
   NOMONITOR
   NOMONITOR var
MONITOR LIST
NEW
PACE=n
PACE=PDL(n)
ROLLIN name
ROLLOUT name
RUN
SELECT
   NOSELECT
SYMBOLS
```

NOTES